PARALLEL PROGRAMMING

(questions from 2020 & 2021 exams – NO oral_exams.zip file)

Davide Giannubilo a.a. 22/23

Parallel algorithms / programming / patterns

(answer with explanation)

Design of parallel algorithms must be performed independently from the architecture which will be used for the implementation: true or false?

During the design of the algorithm the answer is yes, we do it independently from the architecture because we do not need to go deep in the details but, during the implementation, we must take care of the architecture because we need to extract the maximum amount of parallelism from that architecture.

Automatic parallelization cannot be used because it produces wrong solutions: true or false?

Automatic parallelization can sometimes produce incorrect solutions, but this is not always the case. Whether automatic parallelization produces correct solutions or not depends on the specific algorithm being parallelized and the implementation of the parallelization. In general, it is important to carefully test any parallelized code to ensure that it produces the correct results.

Why data scalability of a program is an advantage? (x2)

Data scalability of a program is interesting because whenever the load of data input increases the program can handle it properly. Allocating more memory, creating more threads, or activating more processing cores, a highly data scalable program is better at not losing performance even though the amount of data increases.
So, more flexibility, a better utilization of resources and improved performance.

What is Bit Level Parallelism and on which type of architecture it can be implemented?

Bit Level Parallelism is a low-level type of parallelism that is typically implemented in hardware. For example, if you compute the bitwise and of 2 integer variables in C, the bit a bit computation is performed in parallel in the sense that the bitwise operations between registers in a CPU happen in parallel.

Any kind of architecture that is able to perform operations between registers in parallel and not only serially can exploit bit level parallelism; the most basic example is single core single issue architecture (SISD).

Execution time is the only metric to be used to evaluate a parallel program: true or false? (x2)

False. The execution time is one of the metrics used to evaluate a parallel algorithm. In fact, there are many other factors that can be important in evaluating the performance and effectiveness of a parallel program, such as the amount of parallelism achieved, the efficiency of the algorithms used, the amount of memory and other resources used, and the scalability of the program across different hardware architectures.

Which type of processing elements can implement Single Instruction Multiple Data parallelism?

In this kind of parallelism, we have a single instruction computed on different data. The execution is synchronous and deterministic, and an example could be the modern GPUs that operate on vectorized elements.

Which are the advantages of using extensions of sequential languages instead of native parallel languages

For a programmer is better to use extensions of sequential languages because they are easier to adopt instead of learning a new language and they can be easily integrated into existing compilers, in fact, the native parallel languages compilers are too immature. But the extensions can describe just some types of parallelism (e.g. no pipeline).

A parallel algorithm is always faster than a sequential algorithm: true or false?

False. There are algorithms that run faster on a sequential implementation and the parallel implementation depends on the architecture on which it is built.

Describe the advantages and disadvantages of MPI

MPI (Message Passing Interface) standard defines the syntax and semantics of library routines that are useful to a wide range of users writing portable message-passing programs in C, C++, and Fortran.

PROs:

- Can be adopted on different types of architectures
- Scalable solutions
- Synchronization and data communication are explicitly managed

CONs:

- Communication can introduce significant overhead
- Programming paradigm more difficult than shared memory-based ones
- Standard does not reflect immediately advances in architecture characteristics

Describe the reduce parallel pattern

It is part of *Collective pattern*. It is used to combine a collection of elements into one summary value. The combiner function needs to be associative and, not necessarily, commutative and combines elements pairwise. An example of combiner functions is the addition or the multiplication.

Imagine we want to compute the sum of the following array $A = [2, 3, 4, 5, 6, 7, 8, 9]$ on a four cores processor. In a sequential way we would sum $2 + 3$, then $5 + 4$ and so on. In a "parallel way", using the reduce pattern, each core will sum elements pairwise at same time, so one core $2 + 3$, the second $4 + 5$ and so on, next two cores will simultaneously compute $5 + 9$ and $13 + 17$, and the last step is to compute $14 + 30$. In this way the sum will be performed in just 3 steps, instead of 7 in the sequential way.

There is a problem with floating point numbers and the order of them because the reduction value can change.

Describe the scan parallel pattern (x2)

The scan parallel pattern is a way of parallelizing a computation in which each iteration of the computation depends on the results of the previous iteration. It is often used for operations like prefix sums, in which each element in an array is replaced by the sum of all the elements that come before it in the array.

In the scan parallel pattern, the input data is first divided into small chunks, with each chunk being processed by a separate computational unit (such as a CPU or GPU). Each chunk is then processed independently, with the result from one chunk being passed on to the next as input. This continues until all the chunks have been processed, at which point the result is produced.

The advantage of the scan parallel pattern is that it allows for efficient parallelization of computations that have a sequential dependency between iterations.

Describe the main features of Apache Spark

Spark provides an interface for programming clusters with implicit data parallelism. It has also an API for different languages and, usually, you can achieve a high level of parallelism using multi-CPUs and distributed memory.

Spark is suitable only for big data applications, but it does not have fully support for GPUs yet.

Main features:

- In-memory computing
- Scalability
- Fault tolerance
- Support for multiple programming language
- Ease of use

Describe the main steps of the evolution of parallel programming technologies

Native parallel programming languages were developed to enable parallel computation but despite how good they were in terms of describing parallel programs they were hardly used because they were difficult to use and to integrate. So libraries and extensions for existing languages were introduced, they had a lot of success because they were easy to use and could be integrated easily with existing compilers. Lately with the advent of cloud computing APIs are more and more used.

Briefly explain the difference between unary and n-ary parallel map

In a unary parallel map, a single input value is mapped to a single output value by a function. This is the most common type of parallel map, and it's often used to apply a function to each element of a list or array in parallel. For example, if you have a list of numbers and you want to compute the square of each number, you could use a unary parallel map to apply the square function to each element of the list in parallel.

In contrast, an n-ary parallel map applies a function to multiple input values at the same time and produces multiple output values as a result. This can be useful when the function being applied has multiple inputs and outputs, and when it's possible to apply the function to multiple input values simultaneously. For example, if you have a list of pairs of numbers and you want to compute the sum of each pair, you could use an n-ary parallel map to apply the sum function to multiple pairs of numbers at the same time.

In general, unary parallel maps are simpler and easier to use than n-ary parallel maps, but n-ary parallel maps can be more efficient when applied to the right problem.

Which are the factors that can contribute to limit the resource scalability of a program?

There are several factors that can limit the resource scalability of a parallel program like:

- The number of available processing cores or threads
- Memory usage
- Communication overhead
- Load imbalance between threads
- Algorithmic complexity

In order to exploit a parallel technology, hardware and compilation tool chain are the only required elements: true or false?

False. In order to exploit a parallel technology, hardware and a compilation tool chain are not the only required elements. In addition to hardware and a compilation tool chain, a parallel program also needs to be written in a way that takes advantage of the available parallelism. This typically involves using a parallel programming model and algorithms that are designed to run efficiently on multiple threads or processes. Without these elements, it may not be possible to fully exploit the parallelism offered by the hardware.

OpenMP

(answer with explanation)

An application which includes OpenMP directives can also be compiled with tools which do not support OpenMP: true or false?

True. The program will be compiled but it will be executed sequentially without achieving any advantages from OpenMP.

How OpenMP can be used to describe Multiple Instruction Multiple Data parallelism?

In OpenMP, the directive `sections` describe the MIMD parallelism of an application. It specifies that the enclosed section(s) of code are to be divided among the threads in the team and executed concurrently.

```
#pragma omp sections [clause…]
{
   #pragma omp section
   { /* code section 1 */ }

   #pragma omp section
   { /* code section 2 */ }
}
```

- `section` is executed once by a thread in the team
- If we want nested `sections`, we need nested `parallel`.

The `nowait` clause applies to the `sections` directive removing the implied barrier at the end.

It is never required to control access to shared variables since the compiler adds the necessary code: true or false?

False. By default, all variables are shared since we are working with a shared memory model but, if I have a shared variable and I do not want that a thread modifies it, I have to add the necessary code in order to avoid that a data is modified concurrently.

Which is the aim of variable scope clauses in OpenMP? (x2)

Since OpenMP is based upon the shared memory programming model, most variables are shared by default. The clauses are used to explicitly define how variables should be scoped and also to define:

- how and which data variables in the *serial* section of the program are transferred to the parallel sections
- which variables will be visible to all threads in the *parallel* sections and which variables will be privately allocated to all threads.

Which is the aim of OpenMP run-time routines? (x2)

The aim is to give to the programmers a variety of functions with which they can control the execution of a program. These functions are contained in the `<omp.h>` header file.

Function	Description
`int` **`omp_get_num_threads`**`()`	Returns the number of threads that are currently in the team executing the parallel region from which it is called.
`int` **`omp_get_thread_num`**`()`	Returns an identifier for the thread making this call. This number will be between 0 and `omp_get_num_threads - 1`. The master thread of the team is thread 0.
`void` **`omp_set_num_threads`** `(int` **`num_threads`**`)`	Sets the number of threads that will be used in the next parallel region.
`double` **`omp_get_wtime`**`()`	Provides a wall clock timing routine, use it in pairs to see how much time passes between calls.
`double` **`omp_get_wtick`**`()`	Returns the precision of the timer used for `omp_get_wtime()`

In OpenMP it is not possible to explicitly synchronize threads: true or false?

False. There is a directive in OpenMP that allow us to synchronize explicitly all threads in the team.

```
#pragma omp barrier
```

When a `barrier` directive is reached, a thread will wait at that point until all other threads have reached that barrier. All threads then resume executing in parallel the code that follows the barrier.

OpenMP supports nested parallelism: true or false?

True. OpenMP supports nested parallelism, which means that a parallel region can be nested inside another parallel region. This allows a program to create multiple levels of parallelism, with each level executing on a different set of threads.

To enable nested parallelism in OpenMP, the `nested` clause must be specified at the parallel directive declaration.

```
#pragma omp parallel nested
{
    // Code executed in parallel

    #pragma omp parallel
    {
        // Code executed in parallel within the outer parallel
region
    }
    // Code executed in parallel
}
```

Which is the aim of the `pragma single`?

It specifies that a section of a code is executed only by a single thread (not necessarily the master thread), the choice on the thread is implementation dependent. There is an implied barrier at the end.

```
#pragma omp single [clauses]
{
   /* single section */
}
```

Which is the difference between `pragma critical` and `pragma single`? (x2)

`pragma single` specifies a section of a code is executed only by a single thread (not necessarily the master thread). Instead, `pragma critical` specifies that code is only be executed on one thread at a time.

The main difference between the `critical` and `single` pragmas in OpenMP is that the `critical` pragma is used to protect shared data from being accessed simultaneously by multiple threads, while the `single` pragma is used to execute a block of code on a single thread, even if it is executed within a parallel region.

In OpenMP directives it is possible to specify the scope of variables. True or false?

Yes, it is possible to specify the scope during the declaration of directives used in the program you are writing.

Which is the aim of the `default` clause in OpenMP?

The aim is to specify the behaviour of un-scoped variables in a parallel region.

```
#pragma omp <name> default (shared | none)
```

The scope of all variables is set to `shared` or `none` (`default(shared)` can be omitted because it is already the default).

Using `none` as a default as a default requires the programmer to explicitly scope all variables using other clauses.

Describe OpenMP `for` directive and its requirement

The `for` directive parallelizes execution of iterations of the cycle. The iterations number cannot be internally modified, and it works properly when there are *no data dependencies* in the loop.

```
#pragma omp for [clauses…]
   <for_loop>
```

The most important clauses are:

- `schedule(type, [chunk])` describes how iterations of the loop are divided among the threads in the team (default schedule is implementation dependent)
- `nowait` removes the implied barrier at the end of each loop chunk execution
- Data scope clauses (`private`, `shared`, `reduction`)

If `parallel` is also specified, clauses can be any clause accepted by the `parallel` or `for` directives, except `nowait`.

schedule clause

```
#pragma omp for schedule(type, [chunk])
```

- `type` can be:
  - `static` → Loop iterations are divided into blocks of size `chunk` and then statically assigned to threads. If `chunk` is not specified, the iterations are evenly divided among threads contiguously
  - `dynamic` → Loop iterations are divided into blocks of size `chunk`, and dynamically scheduled among threads. The default chunk size is 1.
  - `runtime` → depends on the environment variable `OMP_SCHEDULE`.
- `[chunk]`:
  - For `dynamic` and `static` must be an integer
  - For `runtime` must be omitted

Which are the clauses which can be associated with `omp for` and which is their aim?

See answer above.

Describe when `omp simd` can be used

The `simd` construct can be applied to a loop to indicate that it can be transformed into a SIMD loop (multiple iterations of the loop can be executed concurrently using SIMD instructions).

```
#pragma omp simd [clause…]
{
   /* for loop */
}
```

If more than one loop is associated with the `simd` construct, then the iterations of all associated loops are collapsed into one larger iteration space that is then executed with SIMD instructions. The sequential execution of the iterations in all associated loops determines the order of the iterations in the collapsed iteration space.

Describe how `omp parallel` directive works

The main thread spawns a team of *slave* threads and becomes the master (thread n. 0 in the within the team). Each thread runs a copy of the code in the block and all thread are *joined* by means of an implied barrier at the end of the block (only the master continues its execution beyond the parallel section).

```
#pragma omp parallel [clauses…]
{
   /* parallel section */
}
```

It is possible to implement nested parallelism after setting a dedicated environment variable.

If any thread terminates within a parallel region, all threads in the team terminate. The work done up to that point is undefined. (Threads may cache their data, flushes should be forced explicitly)

Some important clauses are:

- `if` → conditional parallelization
- `num_threads(int)` → number of threads to spawn
- those related to data scope, reduction, etc.

What does it happen if `pragma parallel` is used without any nested directive? (x2)

Well, since there is no nested directive, each thread will run a copy of the code in the block (the natural behaviour of the directive).

OpenMP and pthread can be used to implement the same type of parallel algorithms. True or false?

True. They are two implementations of a shared memory parallel programming model with threads.

Pthread

(answer with explanation)

Describe pthread condition variable and where they are used

Condition variables allow threads to synchronize explicitly by signalling the meeting of a condition (without condition variables, the programmer would need to poll to check if the condition is met). The condition variables are used beside mutex variables in a synchronization context between threads.

```
int pthread_cond_wait(pthread_cond_t *cond,
                      pthread_mutex_t *mutex)
int pthread_cond_broadcast(pthread_cond_t *cond)
int pthread_cond_signal(pthread_cond_t *cond)
```

When a thread wants to wait for a condition to be met, it first acquires the mutex lock associated with that condition. This ensures that the thread has exclusive access to the shared data that it needs to check the condition. Once the thread has acquired the lock, it can then wait on the condition variable. This causes the thread to release the mutex and enter a dormant state until the condition is signalled by another thread.

When the condition is signalled, the thread that was waiting on the condition variable is awakened and re-acquires the mutex lock. At this point, the thread can check the condition to see if it has been met and proceed accordingly. If the condition has not yet been met, the thread can choose to wait on the condition variable again, or to continue executing.

What is a mutex variable in pthread? (x2)

Mutex (mutual exclusion) variables are the basic method to protect shared data when multiple writes occur.

```
int pthread_mutex_lock(pthread_mutex_t *mutex)
int pthread_mutex_trylock(pthread_mutex_t *mutex)
int pthread_mutex_unlock(pthread_mutex_t *mutex)
```

If it is not locked by another thread, only one thread can lock a mutex variable at any given time. If several threads try to lock a mutex, only one will be successful. Threads that try to acquire a locked mutex are blocked and will try to acquire it as soon as it is released.

Multiple mutexes locking can lead to *deadlocks*.

There are three types of mutex variables depending on their behaviour with respect to deadlocks:

- Normal
- Recursive
- Error check

Describe how it is possible to pass arguments of whatever type to threads created by `pthread_create` (x2)

The `pthread_create()` function starts a new thread in the calling process. The new thread starts execution by invoking `start_routine()`.

```
int pthread_create(pthread_t *restrict thread,
                   const pthread_attr_t *restrict attr,
                   void *(*start_routine)(void *),
                   void *arg)
```

- `thread` → identifier for the new thread returned by the subroutine
- `attr` → used to set thread attributes
  - Joinable (return and exit status) or detached
  - Scheduling
  - Stack size
- `start_routine` → the C routine that the thread will execute once it is created.
- `arg` → argument passed to `start_routine`. It must be passed by address as a pointer cast of type void.

To pass arguments of whatever type we can use a void pointer, which can hold a reference to any type of data. This can be done by defining a function that takes a void pointer as its argument, and then casting the argument to the desired data type within the function.

Threads created in a pthread application are hierarchically organized: True or False?

False. There is no dependency between threads. Once a thread is created, it is a peer and it may create other thread(s). There is no father-child dependency.
The maximum number of threads depends on particular implementation.

Describe how `pthread_join` works

Threads can be created as joinable or detached and a joinable thread can become detached but not vice versa (by default it should be joinable).

```
int pthread_join(pthread_t thread, void **retval)
```

The `pthread_join()` function waits for the thread specified by thread to terminate. If that thread has already terminated, then `pthread_join()` returns immediately. The thread specified by thread must be joinable.
If `retval` is not NULL, then `pthread_join()` copies the exit status of the target thread.
If the target thread was cancelled, then `PTHREAD_CANCELED` is placed in the location pointed to by `retval`.
If multiple threads simultaneously try to join with the same thread, the results are undefined. If the thread calling `pthread_join()` is cancelled, then the target thread will remain joinable.

On success, `pthread_join()` returns 0; on error, it returns an error number.

EXTRA

(topics not present in previous questions)

Types of parallelism

Bit Level Parallelism

Bit-level parallelism is a form of parallel computing based on increasing processor word size. Increasing the word size reduces the number of instructions the processor must execute in order to perform an operation on variables whose sizes are greater than the length of the word.
It is very relevant in hardware implementation of algorithms.

Instruction Level Parallelism

Instruction-level parallelism (ILP) is the parallel or simultaneous execution of a sequence of instructions in a computer program. More specifically ILP refers to the average number of instructions run per step of this parallel execution.

Task Level Parallelism

Task parallelism is a form of parallelization of computer code across multiple processors in parallel computing environments. Task parallelism focuses on distributing tasks, concurrently performed by processes or threads, across different processors.

Parallel pattern – c

(pattern-c.pdf)

Gather pattern

Gather pattern creates an output array starting from a source collection of data and an array of positions in which the values have to be written.

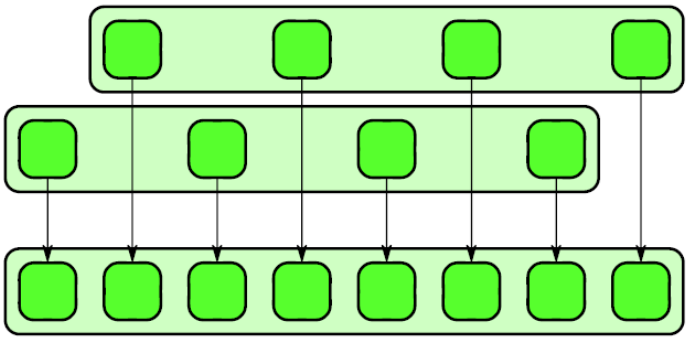
input array =	[3 7 0 1 4 0 0 4 5 3 1 0]
indices array =	[1 9 6 9 3]
output array =	[7 3 0 3 1]

So, the output values are the input values chosen following the indices order. An important thing is that the output array has the *same dimension* of the array of indices and there no operations on the values, everything is the same.

(can be considered a combination of map and random serial read operations)

A *special case* of gather is **shifts**. It moves the date to the left or to the right and the movements Shifts can be handled efficiently with vector instructions because of regularity, and they can also take advantage of good data locality.

Another special case of gather is **zip**. It simply interleaves data like a zipper.



The last special case **unzip**. It simply reverses the zip function, so extracts sub-arrays at certain offsets and strides from an input array.

Scatter pattern

Scatter pattern is the contrary of gather pattern. It uses a collection of input data and a collection of write locations and writes the values in the output array.

input array =	[3 7 0 1 4 0 0 4 5 3 1 0]
indices array =	[2 4 1 5 5 0 4 2 1 2 1 4]
output array =	[0 1 3 0 4]

In this pattern we could have some collisions like when we have to compute the output value for index 1.
How to solve these collisions/race conditions?

Atomic scatter → non-deterministic approach; upon a collision there is no rule that determines which of the input items will be retained.

Permutation scatter → this simply states that collision are illegal; the output is a permutation of the input.

Merge scatter → when you have a collision, there is a commutative and associative operator provided that merges the values. (associative and commutative properties are required since scatters to a particular location could occur in any order)

Priority scatter → every element in the input array is assigned a priority based on its position; priority is used to decide which element is written in case of a collision.

Pack pattern

It is used to eliminate unused elements from a collection. It uses an input array and an array of binary values that represents which values need to be moved into the output array.

input array =	[A B C D E F G H]
binary array =	[0 1 1 0 0 1 1 1]
output array =	[B C F G H]

The **unpack** is the opposite. Given the same data on which elements were kept and which were discarded, spread elements back in their original locations.

Generalization of Pack: Split

Given the same data of pack pattern, so input array and binary array, it moves up or down the elements that have the same value in the binary array.

input array =	[A B C D E F G H]
binary array =	[0 1 1 0 0 1 1 1]
output array =	[B C F G H A D E]

The important thing is that it does not lose information like pack.

Generalization of Pack: Unsplit

It is the inverse of split. It creates an output array based on original input collection.

Generalization of Pack: Bin

It is a split that supports more categories.

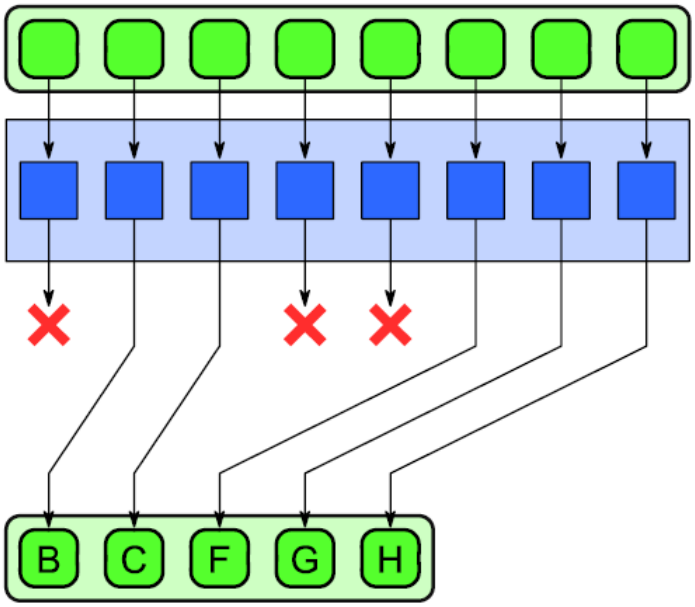
input array =	[A B C D E F G H]
binary array =	[2 0 1 3 3 1 1 2]
output array =	[B C F G A H D E]

In this case four categories: 0, 1, 2, 3.

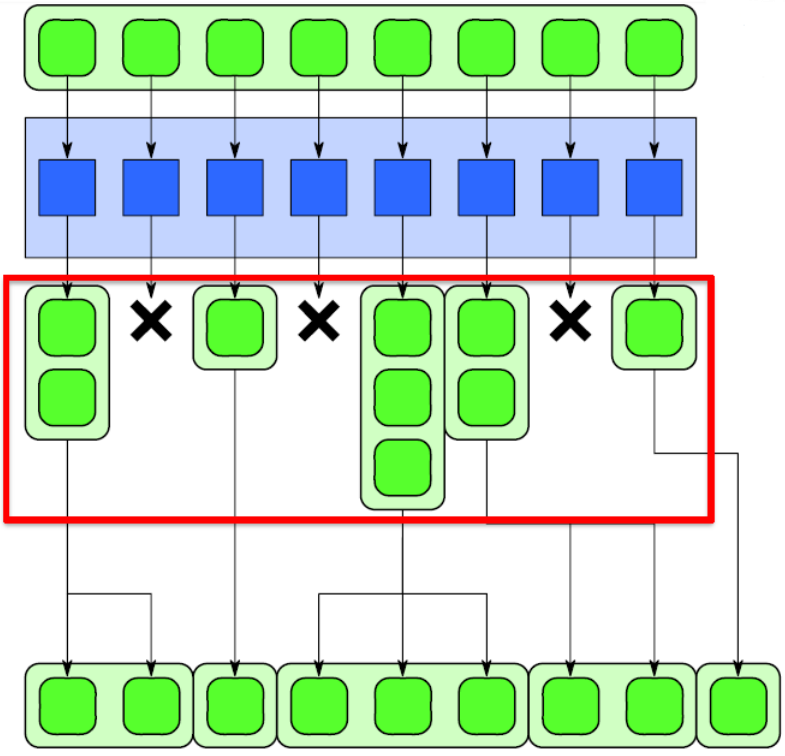
Fusion of Map and Pack

It worth it if most of the elements of a map are discarded.

Map checks for collision
Pack stores only actual collision



Generalization of Pack: Expand



Partitioning Data

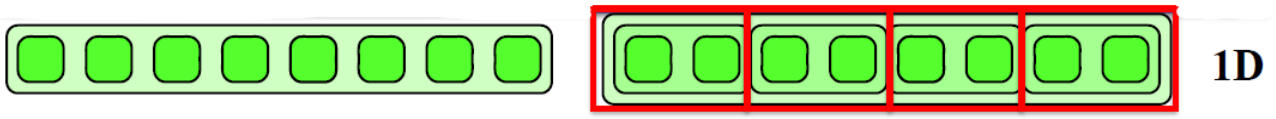
Common strategy:

1) Divide up the computational domain into sections
2) Work on the sections individually
3) Combine the results

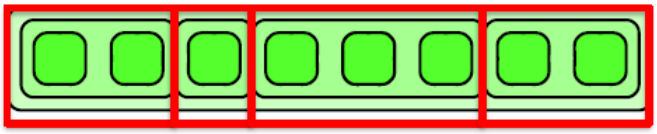
Methods:

- Divide and conquer
- Fork join
- Geometric decomposition
- Partitions
- Segments

Partitioning
Data divided into non-overlapping and equal-sizing regions.



Segmentation
Data is divided into non-uniform non-overlapping regions.



Array of Structures vs Structures of Arrays

The first one may lead to better cache utilization if data is accessed randomly.

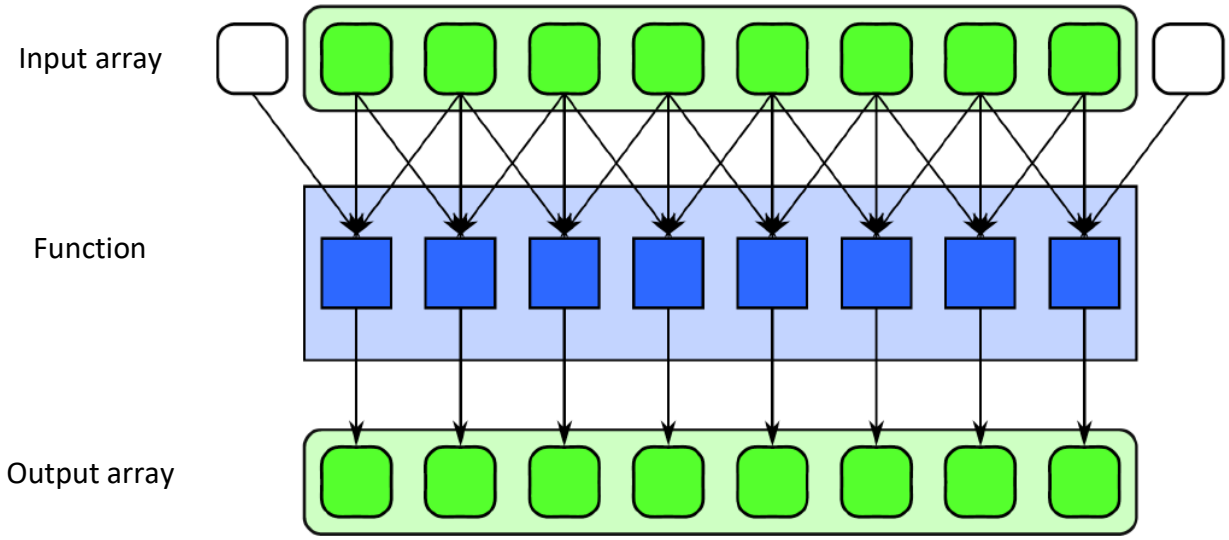
The second one is typically better for vectorization and avoidance of false sharing.

Parallel pattern – d

(pattern-d.pdf)

Stencil pattern

A stencil pattern is a map where each output depends on a *neighbourhood* of inputs. These inputs are a set of fixed offsets relative to the output position. The output is a function of a neighbourhood of elements in an input collection.



Stencil pattern can also be applied on one dimensional and multidimensional data.

Imagine having a matrix and we are pointing an element *P*, its neighbourhood could be the 4 adjacent cells (4-point stencil), the 4 adjacent cells and itself (5-point stencil) or the 8 cell around and itself (9-point stencil).

1) Starting from the first element, 9, averages itself with and the 4 adjacent cells like below
2) Store result in the output
3) Repeat 1 and 2 for all numbers (except the 0s)

0	0	0	0
0	9	7	0
0	6	4	0
0	0	0	0

0	0	0	0
0	4.4	4.0	0
0	3.8	3.4	0
0	0	0	0

An alternative to this implementation is to modify directly the input array without using an output array. In this way, we will calculate the next value using the result of the previous one.

0	0	0	0
0	9	7	0
0	6	4	0
0	0	0	0

0	0	0	0
0	4.4	3.08	0
0	2.88	1.992	0
0	0	0	0

When you are pointing *7* in the input, the “output” will be done using the result of the calculation made for the previous element.

Which result is correct? "*Depends on the implementation*".

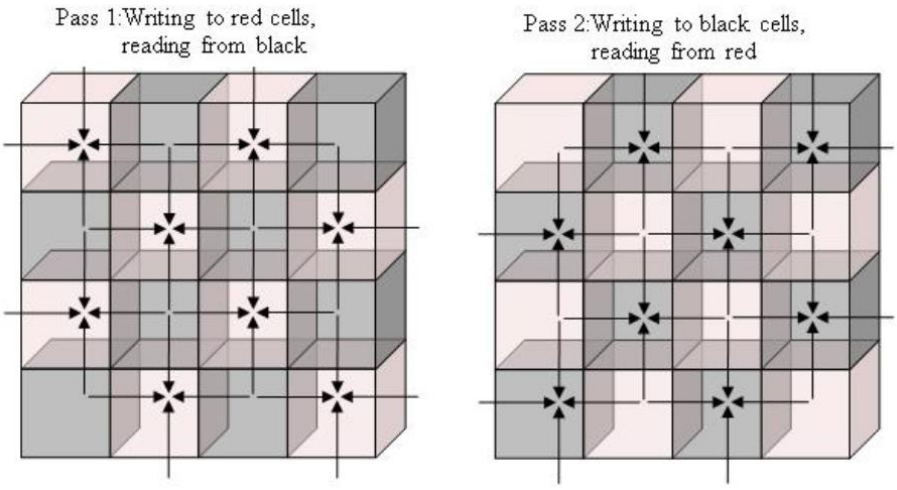
Iterative codes are ones that update their data in steps. Stencils essentially define which elements are used in the update formula and because the data is organized in a regular manner, it can be applied across the data uniformly.

Jacobi iteration

- Consider a 2D array of elements
- Initialize each array element to some value
- At each step, update each array element to the arithmetic mean of its N, S, E, W neighbours
- Iterate until array values converge
- Here we are using a 4 points stencil

Successive Over Relaxation (SOR)

SOR is an alternate method of solving partial differential equations. While the Jacobi iteration scheme is very simply and parallelizable, its slow convergent rate renders it impractical for any real-world applications.
One way to speed up the convergent rate would be to *over predict* the new solution by linear extrapolation. It also allows a method known as Red Black SOR to be used to enable parallel updates in place.

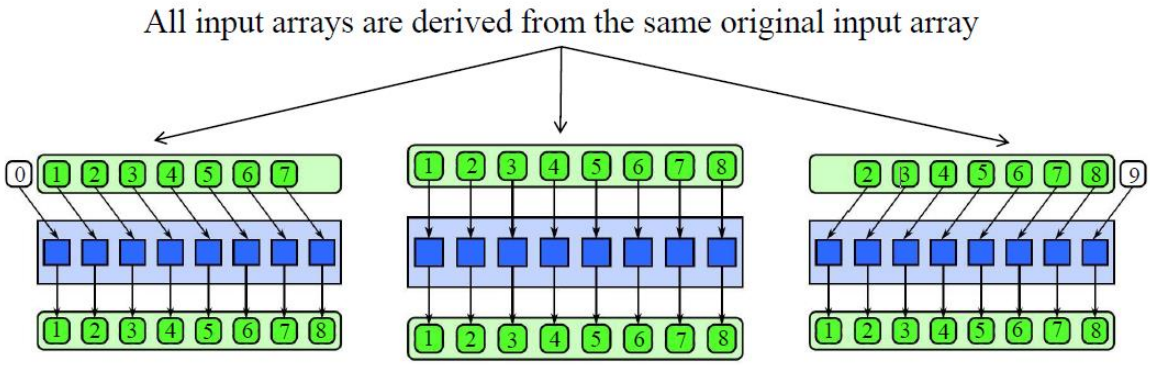


Stencil with shifts

One possible implementation of the stencil pattern includes shifting the input data.
For each offset in the stencil, we gather a new input vector by shifting the original input by the offset amount.

This implementation is only beneficial for one dimensional stencil or the memory-contiguous dimension of a multidimensional stencil.

Memory traffic to external memory is not reduced with shifts but, they allow vectorization of the data reads, which may reduce the total number of instructions.



Stencil and cache optimizations

Assuming 2D array where rows are contiguous in memory.

- Horizontally related data will tend to belong to the same cache line
- Vertical offset accesses will most likely result in cache misses

Assigning rows to cores:

- Maximizes horizontal data locality
- Assuming vertical offsets in stencil, this will create redundant reads of adjacent rows from each core

Assigning columns to cores:

- Redundantly read data from same cache line
- Create false sharing as cores write to same cache line

Assigning *strips* to each core can be a better solution:

- *strip-mining* → an optimization in a stencil computation that groups elements in a way that avoids redundant memory accesses and aligns memory accesses with cache lines
- A strip's size is a multiple of a cache line in width, and the height of the 2D array
- Strip widths are in increments of the cache line size so as to avoid false sharing and redundant reads
- Each strip is processed serially from top to bottom within each core

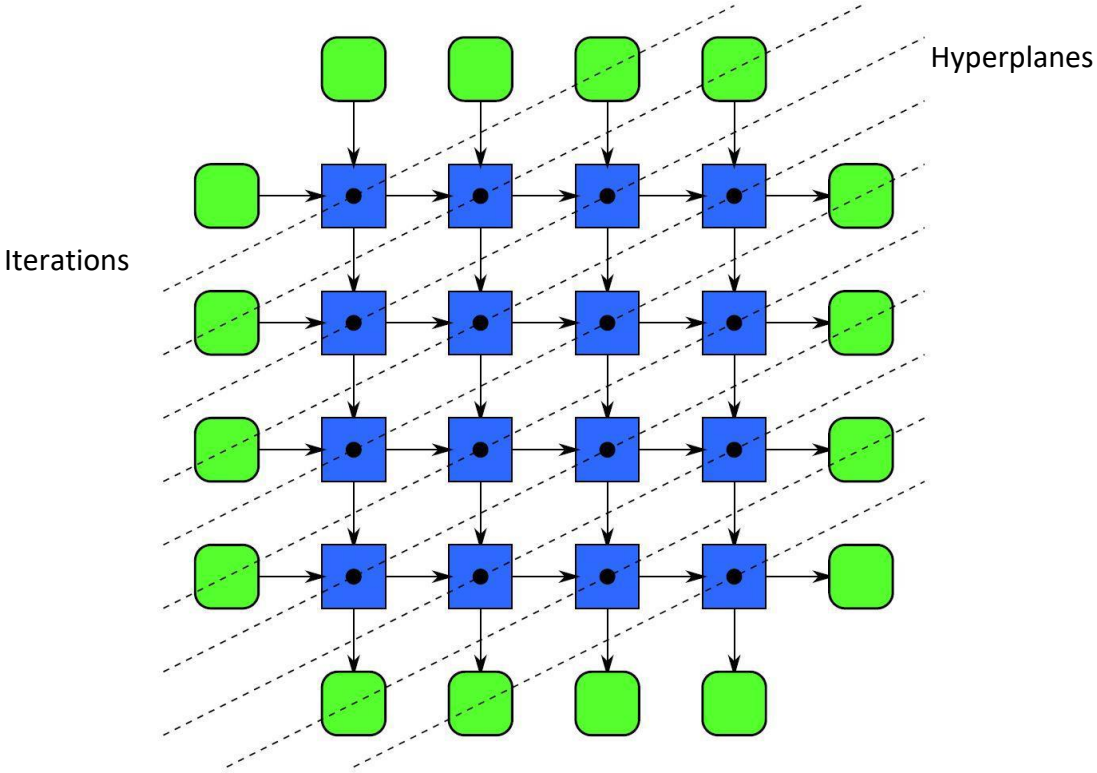
Recurrence

What if we have several nested loops with data dependencies between them when doing a stencil computation?

Find a plane that cuts through grid of intermediate results

- Previously computed values on one side of plane
- Values to still be computed on other side of plane
- Computation proceeds perpendicular to plane through time (this is known as a sweep)

This plane is called a *separating hyperplane*.


Hyperplanes
Iterations

MPI (Message Passing Interface)

1. Introduction

MPI is a library-based specification for parallel and distributed computing developed by the MPI Forum. This single standard is supported by multiple implementations from multiple groups and vendors, in a similar fashion to OpenMP.

It is specific for *Fortran* and *C* but it offers binding for many other languages.

MPI is designed for explicitly implementing parallelism in distributed memory environments with processes, using a library-based syntax. Implementations support many parallel architectures, not necessarily distributed and compilation tools are typically wrappers that set the required flags and environment on top of existing compilers.

Each MPI process operates on its own environment.

- Messages can be instructions, data or synchronization signals
- Delays introduced by communication are much larger than the ones used in shared memory models

MPI provides also:

- buffer management API in order to create and destroy message buffers and to map processes to ranks
- message passing API in order to define the communication and apply some technique like reduction, gathering and scattering

2. Initialization of the system

MPI_Init

```
int MPI_Init(int *argc_p, char ***argv_p)
```

- **NO** MPI routine can called before this one
- `argc_p` and `argv_p` → pointers to the main function `argc` and `argv` respectively
- Sets up the storage for message buffers and ranks the processes of the program
- It can accept `nullptr` arguments
- If there is an error, it will return an error code (like most MPI routines) or `MPI_SUCCESS`
- In multithreaded programs `MPI_Init_thread` should be used instead

MPI_Finalize

```
int MPI_Finalize(void)
```

- **NO** MPI routines can be called after this one (including another `MPI_init`)
- De allocates the message buffers and cleans up all MPI states
- Must be called only when there are no pending messages or unreceived messages

3. Point to Point communication

Processes can be addressed via communicators. A communicator is a collection of processes that can send messages to each other, and we can define custom communicators.

The communication can be:

- Collective → default communicator includes all the processes created when launching the program
- Point to point → requires the user to explicitly address which process inside a communicator must be reached by the message

MPI_Comm_size & MPI_Comm_rank

```
int MPI_Comm_size(MPI_Comm comm, int *size)
int MPI_Comm_rank(MPI_Comm comm, int *rank)
```

- `size` is the number of processes it collects and allows to reach
- `rank` is the number, between [0, size), with which every process is identified within a communicator
- The first argument is the communicator defined using the special type `MPI_Conn`
- The return values are stored in the second arguments of the functions since the actual return value is used as an error code

MPI_Send

```
int MPI_Send(const void *buf, int count,
             MPI_Datatype datatype,
             int dest, int tag,
             MPI_Comm comm)
```

- `MPI_Send` is a routine used to put a message in a communicator buffer where it should be read but the destination process
- This is a *blocking* routine; it will not return until the destination process reads the message

MPI_Recv

```
int MPI_Recv(const void buf, int count,
             MPI_Datatype datatype,
             int source, int tag,
             MPI_Comm comm,
             MPI_Status *status)
```

- `MPI_Recv` is used to read a message from a communicator
- Also this routine is a *blocking* one so, it will not return until the source target has written something in the buffer
- If an `MPI_Recv` has no corresponding send, the process will be blocked forever or worse match another send
- `buf` is the array storing the data to send or ready to receive data

- `count` states how many replicas of the data type will be sent, or the maximum allowed in the buffer
- `source` and `dest` are ranks identifying the target sender or receiver
- `tag` is used to distinguish messages travelling on the same connection (may be ignored most times). It is a nonnegative int and can be used to distinguish messages that are otherwise identical.
    - Processes can use different tags to encode different kinds of messages in the communication
    - A receive can accept any tag using `MPI_ANY_TAG` in the tag field
- `status` provides detailed information on received data. The array size is `MPI_STATUS_SIZE`:
    - `status(MPI_SOURCE)` rank of the sender, useful when sender is `MPI_ANY_SOURCE`
    - `status(MPI_TAG)` tag of the message, useful when message tag is `MPI_ANY_TAG`

Sending and receiving messages introduces a *synchronization* element between processes and where there is synchronization, there are also *deadlocks*.
Two approaches to prevent them:

- Either smartly rearrange the communications
- Use non-blocking sending and receiving routines (`MPI_Isend` and `MPI_Irecv`, but then you will need to also use `MPI_Wait` on the generated request handle to check that all communications have completed before finalizing)

MPI need to know what kind of messages it is delivering and since it cannot use C/C++ datatypes, MPI defines its *own datatypes*.

MPI datatypes		
MPI_CHAR	MPI_UNSIGNED_CHAR	MPI_FLOAT
MPI_SHORT	MPI_UNSIGNED_SHORT	MPI_DOUBLE
MPI_INT	MPI_UNSIGNED	MPI_LONG_DOUBLE
MPI_LONG	MPI_UNSIGNED_LONG	MPI_BYTE

In MPI, if process *q* sends two messages to process *r*, then the first message sent by *q* must be available to *r* before the second message but there is *no restriction* on the arrival of messages sent from different processes. In fact, if *q* and *t* both send messages to *r*, then even if *q* sends its message before *t* sends its message, there is no guarantee that *q*'s message becomes available to *r* before *t*'s message.

4. Output & Input

Typically, process 0 is in charge of writing to the *standard output* but MPI standard does not specify which processes have access to which I/O devices.
Virtually all MPI implementations allow all the processes in `MPI_COMM_WORLD` full access to standard output and error but output is random.

Unlike output, most MPI implementations only allow process 0 in `MPI_COMM_WORLD` access to standard input. The common practice is that process 0 performs `std::cin` and then it broadcasts, or scatters, input values to all processes and it is possible to allow all processes to read from stdin using `-stdin all` when running the program.

5. Compile & Run

MPI is library based, but the flags to the compiler required to make it work are not trivial.

To compile MPI code a compiler wrapper is typically used:

```
mpicxx file1.cpp [file2.cpp …] -o exe
```

To run MPI executables, a dedicated launcher is required:

```
mpiexec -np 4 exe
```

6. Collective communication

All collective communications involve all processes in a communicator, otherwise they result in a *deadlock*; collective communications do not need tags for messages, and they are all ***blocking routines***. Their use is generally more efficient (and simpler) than using point to point communication.

MPI_Bcast

```
int MPI_Bcast(void *buffer,
              int count,
              MPI_Datatype datatype,
              int root,
              MPI_Comm comm)
```

- Delivers data from the local `buffer` of the `root` process to the local `buffers` of all processes
- `root` is the rank of data source
- There is no field to specify a tag
- The same invocation can result in data sending or collection, depending on the process rank in the communicator

MPI_Reduce

```
int MPI_Reduce(const void *sendbuf,
               void *recvbuf, int count,
               MPI_Datatype datatype,
               MPI_Op op, int dest,
               MPI_Comm comm)
```

- Compute a reduction operation defined in `op` on data sent from all processes in `sendbuf`
- Store the result in `recvbuf` locally to process with rank `dest`

- If `count > 1`, `sendbuf` and `recvbuf` are treated as arrays of `datatype` elements and the reduction is performed elementwise on the array

MPI_Op	Operation
MPI_MAX	Maximum
MPI_MIN	Minimum
MPI_SUM	Sum
MPI_PROD	Product
MPI_LAND	Logical AND
MPI_BAND	Bitwise AND
MPI_LOR	Logical OR
MPI_BOR	Bit-wise OR
MPI_LXOR	Logical XOR
MPI_BXOR	Bitwise XOR
MPI_MAXLOC	Maximum value and location
MPI_MINLOC	Minimum value and location

MPI_Allreduce

```
int MPI_Allreduce(const void *sendbuf,
                  void *recvbuf, int count,
                  MPI_Datatype datatype,
                  MPI_Op op,
                  MPI_Comm comm)
```

- The result of the reduction is stored in the `recvbuf` of all processes in the communicator (reduce + broadcast operation)
- These operations can involve a lot of data, having two different buffers may hamper the performance
- Using `MPI_IN_PLACE` in place of a `sendbuf` enables the use of a single buffer for both input and output

MPI_Scatter

```
int MPI_Scatter(const void *sendbuf,
                int sendcount, MPI_Datatype sendtype,
                void *recvbuf, int recvcount,
                MPI_Datatype recvtype,
                int root, MPI_Comm comm)
```

- Scatters data evenly across all processes in the communicator
- `sendcount` is the amount of element to be stored in each `recvbuf` structure
- As for the previous routines, the `sendbuf` can be a `nullptr` when the process is not the root process

MPI_Scatterv

```
int MPI_Scatterv(const void *sendbuf,
                 const int sendcounts[], const int displs[],
                 MPI_Datatype sendtype,
                 void *recvbuf, int recvcount,
                 MPI_Datatype recvtype,
                 int root, MPI_Comm comm)
```

- Scatters data across all processes according to a user-defined distribution
- Process with rank k will receive `sendcounts[k]` elements, starting from the position `displs[k]` of the `sendbuf`

MPI_Gather

```
int MPI_Gather(const void *sendbuf,
               int sendcount, MPI_Datatype sendtype,
               void *recvbuf, int recvcount,
               MPI_Datatype recvtype,
               int dest, MPI_Comm comm)
```

- Join *even* chunks of data from all processes in a communicator.
- The result stored in `recvbuf` is a single vector of elements in `dest` process, with `size = communicator size * recvcount`
- There are additional functions extending this functionality with already seen characteristics, e.g. `MPI_Allgather`, `MPI_Gatherv`, `MPI_Allgatherv`, …

MPI_Barrier & MPI_Wtime

```
int MPI_Barrier(MPI_Comm comm)
```

- Explicit synchronization for all processes in a communicator

```
double MPI_Wtime(void)
```

- Provides a processor dependent time measurement to evaluate the computing time of a portion of program
- The times provided may be processor independent if `MPI_WTIME_IS_GLOBAL` env variable is defined as true

MPI_Comm_split (Create a communicator)

```
int MPI_Comm_split(MPI_Comm comm, int color,
                   int key, MPI_Comm *newcomm)
```

- Partitions the group associated with comm into disjoint subgroups, one for each value of `color`
- Ranks are assigned, per each new communicator, according to the value specified in `key`

- A process may supply the colour `MPI_UNDEFINED`, in which case `newcomm` returns `MPI_COMM_NULL`

Halide

Nowadays we are surrounded by data-intensive imaging applications. These applications require performance and power to process every image. The actual solutions are built using C++ with multithreading and SIMD operations, or specific framework for GPUs like CUDA and OpenCL, or optimized libraries like OpenCV. But writing efficient image processing pipelines is hard and the optimization requires manually transformation of the program and the data structure to reach a good level of parallelism.

Halide proposes a different approach: decouple the algorithm from the schedule.

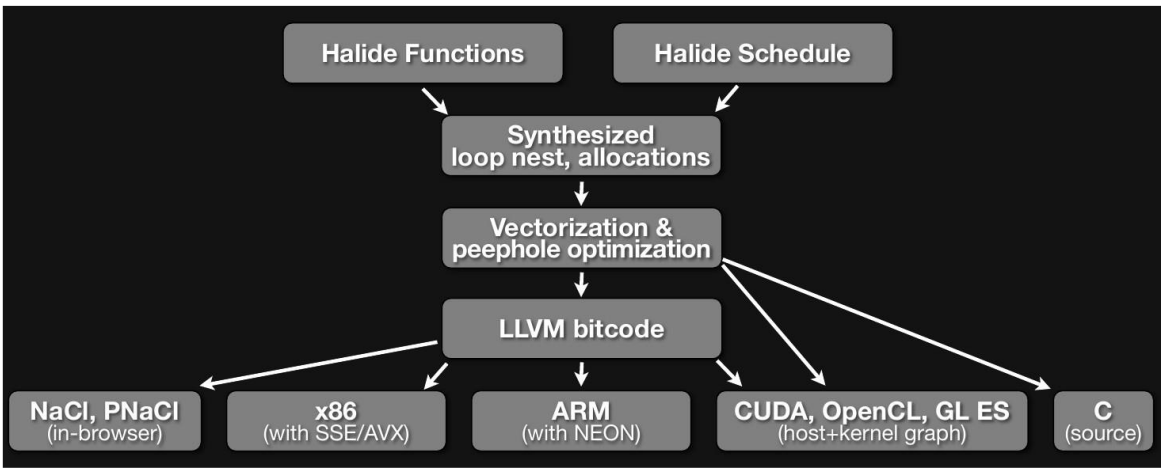
- algorithm → what is computed → fixed once and never modified by scheduling decisions
- schedule → where and when it is computed → easily changed to explore different possibilities

With Halide, changing the schedule does not require any changes to the algorithm and this allows the programmer to experiment with scheduling and finding the most efficient one.

Halide is a *Domain Specific Language* embedded in C++, plus a specialized compiler

- pipelines can be compiled and run just-in-time (evaluate functionality immediately) or they can be statically compiled to an object file and header to be linked later in another program

Development started at MIT, now it is open source and widely used in production by Google, Adobe and other companies.

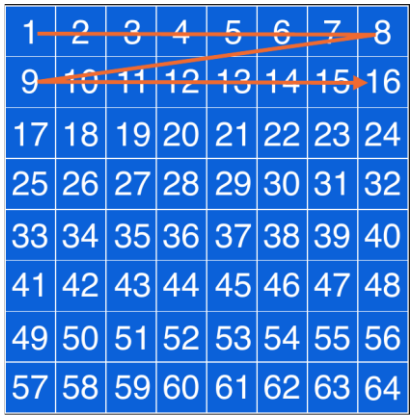


There are some limitations:

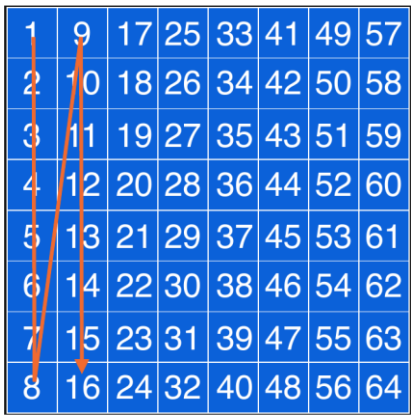
- all computations are over regular grids
- only feed-forward pipelines
- recursions must have bounded depth
- schedule must be specified manually (exception: autoscheduler)

Of course, in the image processing pipeline, we can choose in which order each stage should compute its inputs and/or when each stage should compute them.

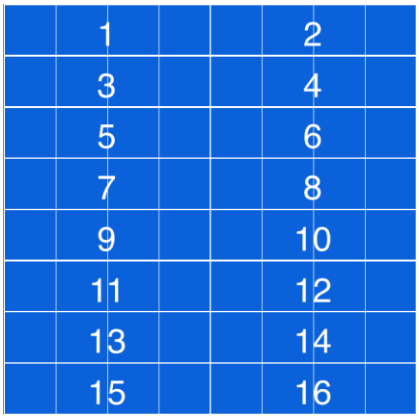
Which order



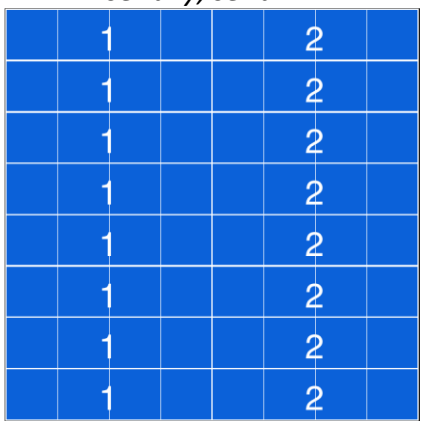
serial y, serial x



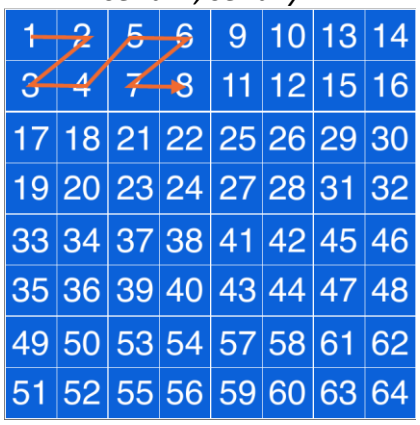
serial x, serial y



serial y, vectorize x by 4

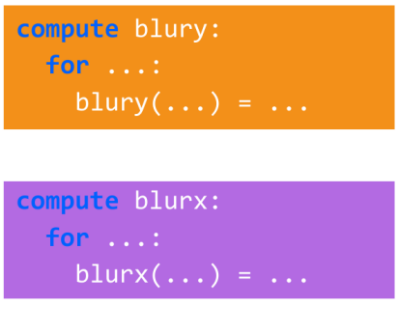


parallel y, vectorize x by 4

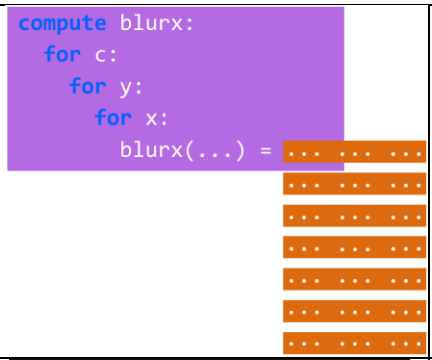


split x by 2, split y by 2
serial youter, serial xouter,
serial yinner, serial xinner

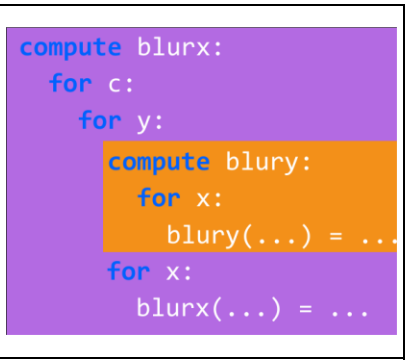
When



Compute and store producer, then compute consumer



Compute producer values when needed by consumer, then throw them away

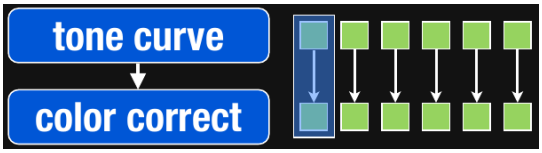


Compute a subset of producer values all at once, then pass them to the consumer

+ no re-computation *- poor locality*	*+ locality* *- redundant work*	*+/- depend on chunk size, kernel size, parallelism*

In the second case, we can keep the old values for reuse and this case we will have locality and no redundant work but less parallelism.

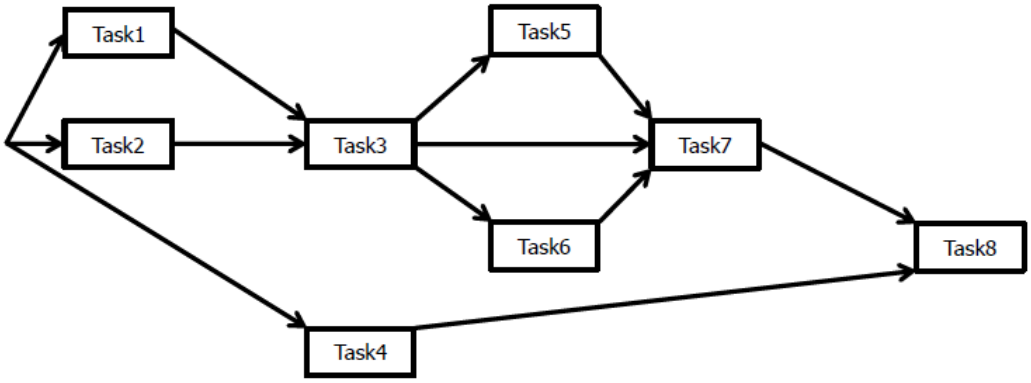
We have also this case where compute producer values when needed by consumer, then throw them away and we obtain more locality and less redundant work.



In the end, performance requires a *complex trade-off* between locality, redundant work and parallelism and a frequent reorganization of computation.

Exercise on task graph

Ex. 1 – Given the graph



and the following values of work: W(Task1) = 1, W(Task2) = 1, W(Task3)=1, W(Task4)=1, W(Task5)= 10 W(Task6)=10, W(Task7)=10, W(Task8)=10.

1) Compute the WORK and the Makespan of the algorithm
2) Ideally which is the minimum number of threads to guarantee the maximum parallelism?
3) Can it be implemented with pthread or OpenMP?

1) The WORK is the sum of all weights → $1 * 4 + 10 * 4 = 44$
   The Makespan is the max weight of the longest path from the start node to the final node → $10 * 3 + 1 * 2 = 32$
2) The number of minimum threads is the ceiling of $\frac{W}{M} \rightarrow \frac{44}{32} = 1.375 \cong 2$
3) Yes, it should be created using both. In OpenMP we can use section pragma to exploit concurrent executions

Ex. 2 – Given the code

```
#include <pthread.h>
#include <stdlib.h>

int data1 = 0;
int data2 = 0;
int data3 = 0;
pthread_t * thread1, * thread2, *thread3;

/*Other global declarations */
void task1(void * input){
data1 = 3;
data2 = 1;
data3 = 2;
}
```

```
void task2(void * input){
data1= data2 + 2;
pthread_create(thread3, &attr, task6, NULL);
}
void task3(void * input){
data2 = data1 + 3;
}
void task4(void * input){
data3 = data1 + 4;
}
void task5(void * input){
data2 = data2 + 5;
}
void task6(void * input){
data1 = data1 + 6;
}
void task7(void * input){
pthread_join(thread2, &return_value);
data3 = data2 + data3;
}
int main(int argc, char ** argv)
{
/* Declaration of data structure and initialization */
pthread_attr_init(&attr);
pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE);
task1();
pthread_create(thread1, &attr, task2, NULL);
pthread_create(thread2, &attr, task4, NULL);
task3();
pthread_create(thread4, &attr, task7, NULL);
pthread_join(thread3, &return_value);
pthread_join(thread4, &return_value);
task5();
printf("%d %d %d\n", data1, data2, data3);
return 0;
}
```

1) Draw the structure (task graph) of the algorithm.
2) List the tasks which can be executed in parallel with task7.
3) Which are the minimum and the maximum values which can be printed by printf (motivate the answer)?

1)

T1	T2 (th1)	T6 (th3)	
	T4 (th2)		T5
	T3	T7 (th4)	

2) The tasks are T6 and T2.
3) Executing the code, I got 8-11-2 | 3-11-2 | 9-11-7. (my guess was 9-11-13) (not sure that is the right answer!)

data1	**data2**	**data3**	
3	1	2	Start
3	1	2	Task2 (data1 = data2+2)
3 | 9	1	2	Task6 (data1 = data1+6)
3 | 9	1	7 | 13	Task4 (data3 = data1+4)
3 | 9	6 | 12	7 | 13	Task3 (data2 = data1+3)
3 | 9	6 | 12	13 | 19 | 25	Task7 (data3 = data2+data3)
I AM NOT SURE ABOUT THESE CALCULATIONS			